

Auto Machine Learning for Early Warning Crisis Detection



Mirko Cesarini, Luca Brusa, Fulvia Pennoni, and Giorgio Vittadini

Abstract In this work, we investigate the application of Automated Machine Learning (Auto ML) techniques to Early Warning Systems (EWSs) for financial crises at the country level. We explore the end-to-end process of applying supervised learning with Auto ML to financial crisis detection, including data preparation, the treatment of imbalanced datasets without relying on synthetic data, and the construction of representative training and testing sets. This study shows how Auto ML can partially automate the selection and tuning of machine learning pipelines (namely, sequences of computational steps, such as preprocessing and classification), thereby reducing the manual effort required while achieving competitive performance. The results indicate that Auto ML provides a valuable starting point for developing financial crisis EWSs, although further manual refinement and ad hoc adjustments of the identified pipelines remain necessary.

Keywords Financial crises prediction · Imbalanced dataset · Longitudinal data · Stratified downsampling

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-032-24141-2_2.

M. Cesarini (✉) · L. Brusa · F. Pennoni · G. Vittadini
University of Milano-Bicocca, Milan, Italy
e-mail: mirko.cesarini@unimib.it

L. Brusa
e-mail: luca.brusa@unimib.it

F. Pennoni
e-mail: fulvia.pennoni@unimib.it

G. Vittadini
e-mail: giorgio.vittadini@unimib.it

1 Introduction

Early Warning Systems (EWSs) are increasingly incorporating Machine Learning (ML) techniques, whose growing popularity in the financial domain (Nazareth & Reddy, 2023) motivates their use for classification, detection, and prediction of financial crises (see also Chap. “[Sampling-Based and Cost-Sensitive Classification in Early Warning Systems for Financial Crises](#)”). However, the application of ML techniques in this context involves several critical issues, e.g., possible class imbalance, where the number of crisis cases is significantly lower than that of non-crises in the available data. It is also important to note that misclassification costs are different according to whether a crisis is missed (false negative) or the EWS issues a false alarm (false positive).

In this chapter, we show an application of Auto Machine Learning (Auto ML) techniques based on supervised learning for EWSs. To the best of our knowledge, such methodology is a novelty for the detection of financial crises. The contribution of this chapter is twofold: on one side, it explores the use of ML and specifically of Auto ML in the context of EWSs, including data preparation, handling of imbalanced data management, and constructing training and testing subsets that accurately represent the original data; on the other side, it demonstrates that many of the tedious tasks involved in tuning ML algorithms can be automated.

This chapter is organized as follows: Sect. 2 presents the state of the art on Auto ML, Sect. 3 describes the dataset used and the preparation steps performed before applying ML algorithms, Sect. 4 shows how Auto ML has been used to find the optimal classification pipeline for financial crisis EWSs, and reports the best pipeline performance. Section 5 draws the conclusions and outlines the future work.

2 State of the Art

In this section, we first survey the ML field, focusing on the Auto ML approaches. Then, we address the issue of imbalanced data management, and we illustrate the Python library which is used to identify the best pipeline for early warning classification.

2.1 Machine Learning

In the following, we propose to use a supervised ML approach, with a binary classifier fitted on a training set to predict upcoming financial crises. Note that a ML algorithm fitted on a specific training set is called a *model* in the ML jargon. Supervised ML is getting very popular in modern data-driven research, enabling predictive modeling across a wide range of scientific domains. Recent advances have focused on improv-

ing generalization, interpretability, and scalability of models, particularly through ensemble methods and deep learning architectures (LeCun et al., 2015; Mahalle et al., 2024).

Shallow learning versus deep learning Shallow learning and deep learning represent two broad paradigms within supervised ML. Shallow learning methods, such as logistic regression, support vector machines, and decision trees, rely on hand-crafted features and typically involve a single or a few layers of transformation from the covariates to the target variable (Hastie et al., 2009). These models are often computationally parsimonious, and are suitable for problems with small datasets or well-understood feature spaces.

In contrast, deep learning (Nielsen, 2015) employs neural networks with multiple layers, enabling automatic extraction of high-level features from raw data and achieving superior performance in complex tasks such as image recognition, natural language processing, and speech analysis (LeCun et al., 2015; Schmidhuber, 2015). However, deep models generally require large labeled datasets and significant computational resources, and they often lack transparency in decision-making (Lipton, 2018). The choice between shallow and deep learning depends on factors such as data availability, problem complexity, interpretability needs, and computational constraints (Domingos, 2012).

In this chapter, due to the peculiarities of the countries' financial crisis domain, the dataset is too small for applying deep learning approaches. Therefore, only shallow learning classifiers are considered.

Tuning machine learning pipelines ML algorithms often include specific parameters whose optimal values cannot be inferred directly from the training data. In the ML jargon, these are called *hyperparameters*, and identifying their optimal values is crucial for classifier performance. In principle, a brute force approach could be adopted to explore all possible hyperparameter combinations: for each set of values, a classifier is fit on a training set, and its performance is then evaluated on a separate dataset. However, this strategy is rarely feasible, as the search space grows exponentially with the number of hyperparameters. Furthermore, many hyperparameters have an infinite domain, e.g., integer and real numbers. Even when a subset of a parameter domain is chosen,¹ a ML pipeline can feature a lot of hyperparameters, and searching the grid of all possible values may be too computationally expensive. The systematic exploration of hyperparameter combinations is known as *grid search* in the ML jargon. Variants such as random search, which samples from a subset of the grid to reduce computational effort, while commonly used, have shown limitations in terms of efficiency and performance (Yaloveha et al., 2022). In short, the issue can be summarized as follows: the larger the grid to explore, the better the configuration that can be identified, but at the cost of exponentially increasing computation time and resources. To address these challenges, alternative approaches have been developed, including the use of Auto ML as discussed in Sect. 2.2.

¹ Focusing on the *C* hyperparameter of the *SVM* *linear* classifier, although its domain ranges on all positive real numbers, frequently the optimal value is searched among a set of powers of 10, e.g., 10^{-3} , 10^{-2} , 10^{-1} , 10^0 , 10^1 , 10^2 , ...

```

1 # Library import
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.linear_model import
   LogisticRegression
5
6 # Pipeline creation
7 pipeline = Pipeline([           # Define the pipeline
8     ('scaler', StandardScaler()), # Step 1:
   Standardize
9                                     # features
10    ('classifier', LogisticRegression()) # Step 2:
   Logit clsf
11 ])
12
13 # Using the pipeline
14 X_train, X_test, y_train, y_test = ...
15 # Fit the pipeline on the training data
16 pipeline.fit(X_train, y_train)
17 # Make predictions
18 y_pred = pipeline.predict(X_test)
19 # ...

```

Listing 1 Example of classification pipeline featuring a standard scaler and a logistic regression classifier

Pipeline In the ML field, pipelines such as the one shown in Listing 1 are frequently used. A pipeline defines a sequence of computational elements, or steps, typically implemented as code modules. Each step processes the input received from the preceding step and produces an output that serves as input for the following step. The first and last elements are exceptions, which respectively process the pipeline's original input data and return the pipeline's output.

In the `scikit-learn` library (Pedregosa et al., 2011), each element in a pipeline has to implement the `fit()` and the `transform()` methods; the only exception is the final element, which is usually a classifier or a regressor, and it must implement the `fit()` and `predict()` methods. The `fit()` method is called when an algorithm is required to learn an internal state from the data that will be used later (e.g., the logit classifier learns once some internal weights during the training, and it will reuse them several times later during prediction); the `transform()` method applies a transformation to the input data (e.g., the `StandardScaler` which standardizes numerical features by removing the mean and scaling to unit variance).

When `fit()` is called on a pipeline, it sequentially calls both the `fit()` method and then the `transform()` method on each element to process the input data, passing the output to the next step. This process continues until the final element (also called *estimator*), where only the `fit()` method is called. During inference, the `predict()` method is called on the pipeline, which calls the `transform()` method on every element of the pipeline except the last one, where the `predict()`

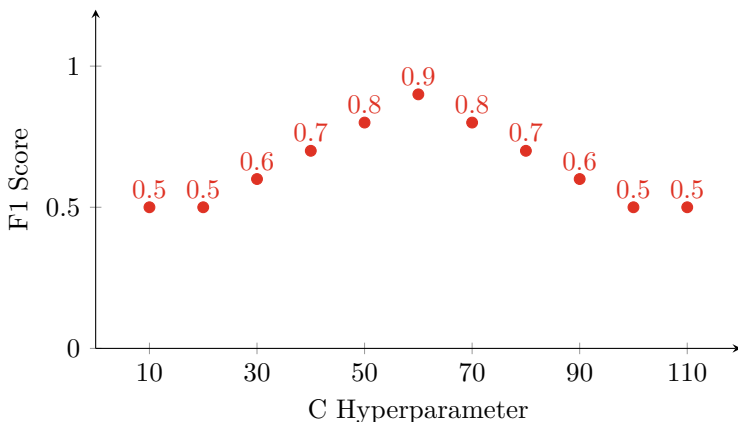


Fig. 1 Hyperparameter values and corresponding F1 score performance plot. Each point represents a specific C value and its corresponding F1 score. It is useful to guide a manual search of the optimal C value that maximizes performance

method is called. This abstraction improves code readability, facilitates reproducibility and parameter tuning. Focusing on the latter, the pipeline hyperparameters can be specified after the pipeline creation, thus facilitating the development of tools that partially or fully automate the hyperparameter tuning process.

2.2 (Auto) Machine Learning

Traditional ML workflows require significant manual effort and domain expertise for tasks such as feature selection, model selection, and hyperparameter tuning. Automated ML has emerged as an approach to automate these phases. The following example illustrates the intuition behind Auto ML: consider the task of exploring values of a hyperparameter C in the range from 1 to 110, to identify the value that maximizes a classifier’s performance. At the beginning, 11 randomly picked values for C are considered, and the corresponding F1 scores² are shown in Fig. 1 to provide evidence on how hyperparameter variation may affect the performance of a classifier.

Based on the preliminary results of Fig. 1, it would be wise to narrow the search space between 40 and 80. Advanced optimization techniques can be used to reduce the effort of systematically exploring the hyperparameter domain space and to identify the optimal model, e.g., Bayesian optimization (Snoek et al., 2012), evolutionary algorithms (Real et al., 2019), and reinforcement learning (Zoph & Le, 2017). Mod-

² The F1 score is the harmonic mean of the precision and recall, with $precision = TP / (TP + FP)$ and $recall = TP / (TP + FN)$. TP, FP, TN, and FN are, respectively, the true positives, false positives, true negatives, and false negatives. It summarizes precision and recall, and it is not affected when precision is improved at the expense of recall or vice versa.

ern Auto ML pipelines typically evaluate components for automatic data cleaning, transformation, model evaluation, and ensemble construction (Feurer et al., 2015).

Although a detailed comparison of the different Auto ML approaches is outside the scope of this chapter, we briefly mention that several open-source and commercial platforms have significantly advanced the accessibility of these techniques. These include Auto-sklearn (Feurer et al., 2015), which integrates meta-learning and ensemble selection, Tree-based Pipeline Optimization Tool (TPOT) (Olson & Moore, 2016), which uses genetic programming for pipeline optimization, AutoKeras (Jin et al., 2019) for deep learning tasks, and cloud-based services such as Google Auto ML (Google, 2025) and Microsoft Azure Auto ML (Azure, 2025).

Recent research has focused on developing more efficient, adaptive Auto ML frameworks. Emerging trends include few-shot and zero-shot Auto ML³ using transfer and meta-learning (He et al., 2021), multi-objective optimization that balances accuracy with latency or interpretability (He et al., 2021), and human-in-the-loop systems that integrate expert feedback to guide search processes more effectively.

Here, we propose the use of TPOT as the Auto ML tool since it explores and optimizes ML pipelines using genetic programming (Olson & Moore, 2016). TPOT provides a predefined set of pipeline components and corresponding hyperparameter search spaces, thereby streamlining the model selection and tuning process without extensive manual intervention (Olson et al., 2016). Its use of genetic programming enables efficient exploration of the search space, often yielding high-performing models with fewer iterations compared to traditional grid or random search approaches (Azedou et al., 2025; Leung et al., 2003). Additionally, TPOT outputs a fully executable Python script (Python Software Foundation, 2025; van Rossum & Drake, 2011) that reproduces the selected pipeline and hyperparameter configuration, enhancing transparency and reproducibility. It also reduces the work of creating and testing the identified pipelines. Importantly, TPOT pipelines are constructed exclusively from components of the `scikit-learn` library, ensuring compatibility with widely used ML standards and facilitating integration into data processing workflows (Pedregosa et al., 2011). Further information about TPOT is provided in Sect. 2.4.

2.3 Imbalanced Data Management

In supervised ML, a dataset is considered imbalanced when instances of one class significantly outnumber those of the other one(s), which poses a challenge to model performance. A classifier trained on an imbalanced dataset tends to be biased toward

³ *Zero (or few) shot classification* refers to the task of predicting a class that was not observed during training (or where only a limited number of examples were provided). This method often involves the use of transfer learning, i.e., a model trained on a different task is adapted for the target task. The terms few-shot and zero-shot classification originated in natural language processing, where large language models and prompting techniques were employed to build zero-shot classifiers (Perez et al., 2021).

the majority class, leading to poor generalization and low predictive performance for the minority class. This issue is particularly critical in domains where the minority class carries high importance, such as fraud detection, medical diagnosis, and anomaly detection (He & Garcia, 2009). Traditional evaluation metrics like accuracy can be misleading in these contexts, as a model may achieve high accuracy simply by predicting the majority class (Chawla et al., 2002). Several techniques exist to address or mitigate the imbalanced data problem, including down-sampling, cost-sensitive learning, and ensemble methods that focus on minority class detection (Krawczyk, 2016).

EWSs based on financial data, like those considered in the present work, are highly imbalanced, since the crisis cases are few compared to the non-crises. Therefore, we focus on down-sampling the majority class by replacing its records with a randomly selected subset, ensuring the new subgroup has the same dimension as the crisis set. In this way, we explore the predictive performance of the ML algorithms on EWSs without adding synthetic data. Ad hoc stratification is used to preserve the data balance between the extracted subset and the original one, as illustrated in Sect. 3.1.

2.4 Tree-Based Pipeline Optimization Tool

TPOT is an open-source Python library that leverages genetic programming to identify and optimize ML pipelines (Olson & Moore, 2016). Built on top of `scikit-learn`,⁴ TPOT automates model selection, feature preprocessing, and hyperparameter tuning using a genetic approach to optimize pipelines.

In TPOT, a pipeline is represented as a tree,⁵ with pipeline elements as nodes, and a set of predefined pipelines is referred to as a *population*. An example of a tree representing a pipeline is shown in Fig. 2.

The genetic algorithm used by TPOT is inspired by Charles Darwin’s theory of evolution (survival of the fittest), in which the fittest individuals have a higher probability of being selected for reproduction. Pipeline elements evolve through operations such as mutation and crossover. The population is evaluated, and only the best-performing elements are passed to the next generation (Mitchell, 1998). TPOT creates the initial population by randomly generating a fixed number of trees, each tree representing a ML pipeline, with elements randomly drawn from a list of classifiers, feature selectors, and other components. TPOT provides some predefined lists (e.g., “TPOT Light”, which restricts the model selection to the shallow learning classifiers available in the `scikit-learn` library) and also allows users to create custom lists. For further details, the interested reader can refer to the *TPOT documentation*

⁴ A Python library widely used in the community (Pedregosa et al., 2011).

⁵ Although each tree node can have only one parent node, this constraint is not respected in TPOT pipelines. Nevertheless, by abuse of notation, TPOT pipelines will be called trees in this work.

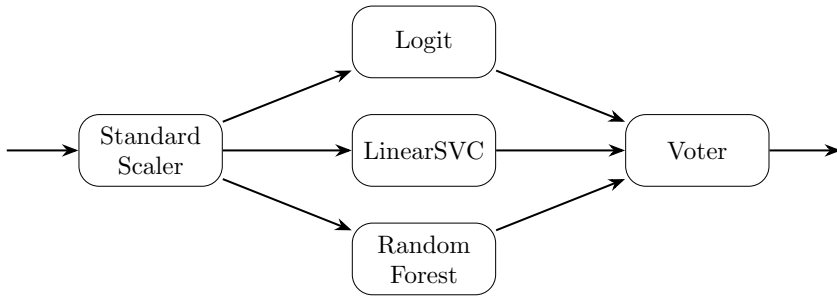


Fig. 2 Example of a pipeline implementing an ensemble classifier: the pipeline starts with a standard scaler followed by three classifiers (logit, linearSVC, and random forest) whose predictions are combined using a Voter. The voter determines the final output based on the majority of the classifier predictions

(EpistasisLab, 2025). Focusing on the next generation, new trees (offspring) are created in a manner analogous to the formation of new chromosomes:

- either by merging two trees from the current generation using the crossover operation (starting from two randomly selected pipelines, both are split at a random point and their contents are swapped);
- or by modifying an element using a mutation operator (inserting, deleting, or randomly replacing an element in the tree).

The new offspring pipelines are checked, and invalid ones are dropped; e.g., a pipeline that attempts to pass a feature set into an element that expects a single integer would be dropped. This can be easily checked because, during the performance evaluation (described next), a malformed pipeline will raise a computation error. Each tree (both the old and the offspring ones) is evaluated using a fitness function, e.g., the classification accuracy (or any available machine learning metric, e.g., F1 score, AUC-ROC, ...) and the pipeline complexity (pipelines using few elements are preferred). Finally, using the evaluation results, a new generation is formed by selecting the fittest pipelines, ensuring that the population size remains constant. Hyperparameters are managed in a similar way to nodes of the tree. Throughout the process, the single best-performing pipeline discovered during the TPOT run is stored separately and serves as the representative pipeline afterward. Through genetic evolution, TPOT can efficiently discover high-performing pipelines that might not be apparent through manual design.

Unfortunately, optimization approaches based on genetic algorithms suffer from limitations such as getting stuck in local minima. Enlarging the available dataset or lengthening the number of generations could mitigate or address the problem. Despite these limitations, Auto ML tools like TPOT are valuable for practitioners seeking effective models without extensive manual tuning. Of course, the pipeline identified by TPOT is not the final step, but often marks the beginning of further research efforts; an example is illustrated in Sect. 4.2.

3 Dataset and Preliminary Preparation

The data considered for the applicative example are the same as those illustrated in Chap. “[Sampling-Based and Cost-Sensitive Classification in Early Warning Systems for Financial Crises](#)”, also used in Chaps. “[Exploring Binary Regression and Hidden Markov Models for Early Warning Systems](#)” and “[A Regularized EWS for Banking Crises: A Grouped Fixed Effects Approach](#)”, and originally presented in Pignini (2021). For the current proposal, they require some adjustments to achieve the best predictive performance with the ML algorithms. In the following, after briefly describing the data, we show how missing values were handled and how additional grouping variables were created to perform stratified downsampling that approximates the original dataset composition as closely as possible. Grouping variables have been built using both data from the original dataset and information from external sources (e.g., country dimensions, geographical location, etc.).

3.1 Data Description

The data used for the application comprise a binary target variable, named `Crisis`, indicating whether a systemic banking crisis occurred across 129 countries over 35 years, from 1983 to 2017. As illustrated in Chap. “[Sampling-Based and Cost-Sensitive Classification in Early Warning Systems for Financial Crises](#)”, a total of 219 crisis events are observed, compared to 2,986 non-crisis instances. As expected, non-crisis episodes significantly outnumber crisis occurrences. A set of macroeconomic, monetary, and financial indicators, each lagged one year relative to the target variable, are included as covariates. These comprise the Real GDP Growth Rate, the Logarithm of the per Capita GDP, the Inflation Rate, the Real Interest Rate, the Ratio of Broad Money to Foreign Exchange Reserves, the Growth Rate of Private Credit, and the Growth Rate of net Foreign Assets to GDP. The latter variable, in particular, contains 160 missing values. These were imputed using the following procedure: missing values were replaced with 0, and a corresponding dummy variable was created (named `Aux`), taking the value 1 for imputed records and 0 otherwise. In addition, we also include the one-year lagged target variable as a covariate.

3.2 Grouping Variables

As previously mentioned, the dataset is highly imbalanced, with a number of crises observed over the considered time span being considerably lower than the number of non-crises. Furthermore, some countries did not experience any crises during the observation period. In this context, using undersampling could result in removing all

available non-crisis data for those countries. This issue can be addressed by grouping countries based on their similarities and ensuring that at least 4 elements (2 crisis and 2 non-crisis records) remain in each group after undersampling. Several grouping strategies have been evaluated; the *acceptance criterion* is described as follows: after applying undersampling and train-test splitting, the two resulting subsets should contain at least one crisis record and one non-crisis record for each group of countries. Grouping was based on the following variables:

- Country geographical information: a variable named *Continent* was created with the following categories: North America and Oceania, South America, Europe, Asia, and Africa. Since the Oceania countries had no crises, this group was merged with the North America group because they share a common language and cultural heritage. In addition, the country's land area plus water bodies (lakes, reservoirs, and rivers), measured in square kilometers (km^2) was considered through a variable named *Land Area Km^2* .
- The *Logarithm of the per Capita GDP*, previously introduced, which focuses on the economic status of the country.

The rationale for focusing on the three variables mentioned above is to group countries based on geographical and economic similarities. Both variables *Land Area Km^2* and *Logarithm of the per Capita GDP* were binned, using the *pandas qcut* function (Qcut, 2025), which discretizes a variable into equal-sized buckets based on quantile cuts computed from the actual values of the variable. Several quantile values were evaluated; the only combination that satisfied the acceptance criterion introduced earlier was two bins per variable. Two additional variables, namely *QR Land Area Km^2* and *QR Log of the per Capita GDP*, were created to store the labels produced by the binning process. These variables were not fed to the ML algorithms; they were used only for downsampling and train-test splitting, as outlined in Sect. 3.3.

3.3 Downsampling and Train-Test Split

A variable named *Sel* was added to the dataset, whose values were computed by concatenating the values of the variables *Continent*, *QR Log of the per Capita GDP*, and *QR Land Area Km^2* . The *Sel* variable was created to support the downsampling and train-test splitting and can be considered as a stratification variable. An example is shown in Table 1. It is worth noting that the records for *Chile* appear in two different *Sel* groups in Table 1. This occurs because the values of the variable *Logarithm of the per Capita GDP*, depending on the year, fall into both bin 0 and bin 1 of the *QR Log of the per Capita GDP* variable.

The dataset was then balanced within each level of the stratification variable *Sel* by undersampling the majority class (non-crises) to match the minority class size (i.e., crises). The resulting records were subsequently split into training and test sets,

Table 1 Example of the `Sel` variable and the other variables used to build it. The `Country Name` was not used to build `Sel`, but it is reported for information purposes only. The records shown are only a subset of the original dataset

Country Name	Continent	QR Log of the per Capita GDP	QR Land Area Km ²	Sel
Burundi	Africa	0	0	Africa.0.0
Cameroon	Africa	1	0	Africa.1.0
South Africa	Africa	1	1	Africa.1.1
Switzerland	Europe	1	0	Europe.1.0
United Kingdom	Europe	1	1	Europe.1.1
Belarus	Europe	0	1	Europe.0.1
Costa Rica	North America and Oceania	1	0	North America and Oceania.1.0
United States	North America and Oceania	1	1	North America and Oceania.1.1
Argentina	South America	0	1	South America.0.1
Chile	South America	0	0	South America.0.0
Chile	South America	1	0	South America.1.0
...	...	...	...	...

while the discarded observations were retained as additional test subsets for potential future use. In this way, the `Sel` variable was used for performing both undersampling and train-test split as outlined in Algorithm 1 in Appendix A. The algorithm labels each record using one of the following:

- `Balanced_Train`: label used for records in the balanced train subset;
- `Balanced_Test`: label used for records in the balanced test subset;
- `Test2`: label used for records not selected for inclusion in the balanced train and test subsets;
- `Test3`: label used for the groups where either the crisis or the non-crisis subset was absent.

Table 2 shows the number of elements in each group identified by the labels described above, stratified by crisis and non-crisis.

Table 2 Number of records in the groups `Balanced_Train`, `Balanced_Test`, `Test2`, and `Test3`. The records dropped during downsampling are all related to the non-crisis class, which is the majority class. All the `Test3` elements actually refer to non-crisis events too

Group label	QT crisis	QT Non-crisis
<code>Balanced_Train</code>	156	156
<code>Balanced_Test</code>	63	63
<code>Test2</code>	–	2603
<code>Test3</code>	–	164

3.4 Rolling Window Dataset

The dataset composed of records from the `Balanced_Train` and `Balanced_Test` described in Sect. 3.3 was used for training and evaluating the ML algorithm, respectively. For reference, this dataset is referred to as `ML_Dataset`. Each record contains information for a single country, with the response variable corresponding to year t and all covariates corresponding to year $t - 1$. Additionally, we considered a dataset implemented using a five-year rolling window strategy, referred to as `ML_Rolling_Dataset` hereafter. In this dataset, each record corresponds to a specific country, at year t for the response variables, while the covariates span over the five preceding years $t - 1, \dots, t - 5$. When the reference year was too close to year 1983, the earliest year in the original data, there were not enough years to fill the rolling window. Therefore, the last available year was repeated until the rolling window reached the desired size.

Missing combinations of country and year were skipped, with an additional previous year included to maintain the window length. It is worth noting that both the `ML_Dataset` and the `ML_Rolling_Dataset` contain the same number of records, each focusing on a specific combination of country and year. The target variable for the optimal pipeline search in both datasets was the feature `Crisis`.

4 Search for the Optimal Classification Pipeline(s)

The search for the optimal pipeline was performed using both the considered datasets (i.e., `ML_Dataset` and `ML_Rolling_Dataset`, introduced in Sect. 3.3), and two pools of elements upon which the `TPOT` library builds the initial pipeline population and starts the search (described in Sect. 4.1). The optimal pipelines found are presented in Sect. 4.2. Further analyses on the best-performing pipeline are reported in Sect. 4.3. Section 4.4 compares the proposed pipelines against baseline models from the literature, while Sect. 4.5 examines feature importance through SHAP values. Some considerations are drawn in Sect. 4.6.

4.1 TPOT Pipeline Search Setup

As previously introduced, TPOT builds the initial population of pipelines by randomly combining the elements drawn from a pool. Two pools of elements were considered:

- **Light_Pool:** This is a pool provided by the TPOT library, which restricts the model selection to the shallow learning classifiers available in the Python library `scikit-learn`. All other elements were unrestricted; e.g., the feature selection elements were all those available from the `scikit-learn` library.
- **Custom_Pool:** This is a custom pool in which the set of classifiers and feature extractors is restricted to the following list. According to the authors' experience, these elements are the most frequently used for tabular data classification. The selected elements, along with their respective Python libraries, are listed below; a note is provided whenever the library element name is not self-explanatory:

- `sklearn.tree.DecisionTreeClassifier`;
- `sklearn.naive_bayes.BernoulliNB`: Naive Bayes classifier for Bernoulli distributions;
- `sklearn.ensemble.ExtraTreesClassifier`: implements a meta estimator that fits several randomized decision trees (also known as extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting;
- `xgboost.XGBClassifier`: eXtreme gradient boosting, is an ensemble learning method that combines multiple weak learners (like decision trees) to create a strong predictive model;
- `sklearn.linear_model.SGDClassifier`: implements regularized linear models with stochastic gradient descent (SGD) learning;
- `sklearn.svm.LinearSVC`: linear support vector classification;
- `sklearn.feature_selection.RFE`: feature ranking with recursive feature elimination;
- `sklearn.feature_selection.SelectFromModel`: a component that selects features based on importance weights learned by an underlying model.

Moreover, each TPOT pipeline search session was started using the following parameters:

- `population_size=100`: the size of the initial population where the genetic optimization is applied;
- `generations=100`: the number of generations; at each generation, new offspring pipelines are created while the less performing pipelines are dropped;
- `random_state=42`: most of TPOT operations are based on random initialization of values. This parameter is used as a seed to make random initialization repeatable;

```

1 from tpot import TPOTClassifier
2
3 # ...
4
5 xTrain=... # The training set covariates
6 yTrain=... # The training set labels
7
8 tpot = TPOTClassifier(
9     generations=100, #Number of iterations
10    population_size=100,
11    scoring='f1_macro', # the F1 score macro
12                      averaged will be used to evaluate the
13                      pipeline performance
14    cv=5, # cross-validation
15    n_jobs=-1, # Use all but one available
16              processors in parallel to expedite
17              computation
18    config_dict='tpot light',
19    random_state=42)
20
21 # The next command starts the optimal pipeline
22   search
23 tpot.fit(xTrain, yTrain)
24
25 # The optimal pipeline is saved in a Python script
26 tpot.export('script.py') # From this script the
27   best pipeline can be extracted

```

Listing 2 Example of code to perform a TPOT optimal pipeline search.

- `cv=5`: when a pipeline is evaluated, 5-folds cross-validation is used on the train set.

An example of TPOT invocation is shown in Listing 2. Once the TPOT pipeline optimization ends, the resulting pipeline is trained from scratch on the train set and evaluated on the test set, as introduced in Sect. 3.3.

An optimal pipeline search takes about 8 h on an Intel I7 Linux machine with 32 GB of RAM.

4.2 TPOT Pipeline Search Results

For each initial search setup, an optimal pipeline was selected, namely P1.C.M, P2.L.M, P3.C.R, and P4.L.R. The suffixes .x.y in P are the following:

- .x refers to the pool of elements upon which the initial pipeline population is built. Possible values are

Table 3 Optimal pipelines selected for each initial search setup, defined by the combination of element pool and dataset, along with the corresponding F1 score

Pipeline nickname	Element pool	Dataset	F1 score
P1.C.M	Custom_Pool	ML_Dataset	0.79
P2.L.M	Light_Pool	ML_Dataset	0.79
P3.C.R	Custom_Pool	ML_Rolling_Dataset	0.86
P4.L.R	Light_Pool	ML_Rolling_Dataset	0.85

The best value of F1 Score is in bold

- **L**, denoting the Light_Pool;
- **C**, denoting the Custom_Pool;
- .y refers to the dataset used during the TPOT search. Possible values are:
 - **M**, denoting the ML_Dataset;
 - **R**, denoting the ML_Rolling_Dataset.

The initial search setup of each selected pipeline is shown in Table 3, as well as the obtained F1 score, while the pipeline description and the performance on the test set are shown next.

Pipeline P1.C.M The pipeline shown in Listing 3 is an evolution of the best pipeline that TPOT identified for the Custom_Pool of elements and ML_Dataset. Sometimes the pipelines can be further improved by applying some changes that a human brain can quickly identify. The original pipeline that TPOT identified is reported in Listing 5 and the description of the manual intervention is outlined in Sect. 5.

In Listing 3, the `make_pipeline` class creates a pipeline instance. Its parameters are the pipeline elements, in the exact order they will occupy the pipeline. The first element is the `StackingEstimator`, a class provided by TPOT for creating pipelines that combine multiple classifiers. It is a meta-transformer that wraps a classifier and produces an output composed of the input dataset enriched by the classifier predictions (and/or class probabilities). When the `StackingEstimator` is created, a classifier is specified and stored internally. The `StackingEstimator` implements the `fit()`, `transform()`, and `predict()` methods required for pipelines, as outlined in Sect. 2.1. During prediction, the internal classifier processes the input features; its predictions are merged with the original input, and the combined result is passed to the next pipeline element. In this case, several classifier outputs are added to the input dataset, namely the `BernoulliNB` and the `DecisionTreeClassifier`.

Finally, the enriched features are given to the `ExtraTreesClassifier`. The pipeline classification report is summarized in Table 4.

Despite the manual intervention, both pipelines (P1.C.M.bis and P1.C.M) have similar performance. Therefore, they will be both referred to as P1.C.M, unless otherwise specified.

```

1 # ...
2 pipe=make_pipeline(
3     StackingEstimator(
4         estimator=make_pipeline(
5             StackingEstimator(
6                 estimator=BernoulliNB(alpha
7                     =10.0, fit_prior=True)),
8                 DecisionTreeClassifier(criterion="
9                     gini", max_depth=6,
10                    min_samples_leaf=11,
11                    min_samples_split=11))),
12     ExtraTreesClassifier(bootstrap=False,
13                          criterion="gini", max_features=0.05,
14                          min_samples_leaf=2, min_samples_split
15                          =13, n_estimators=100))

```

Listing 3 The P1.C.M.Bis Pipeline Code

Table 4 Classification report of the pipeline P1.C.M.Bis evaluated on the test set

	Precision	Recall	F1 score	Support
Non-crises (0)	0.76	0.83	0.79	63
Crises (1)	0.81	0.75	0.78	63
Accuracy	–	–	0.79	126
Average	0.79	0.79	0.79	126

Table 5 Classification report of the pipeline P3.C.R evaluated on the test set

	Precision	Recall	F1 score	Support
Non-crises (0)	0.80	0.95	0.87	63
Crises (1)	0.94	0.76	0.84	63
Accuracy	–	–	0.86	126
Average	0.87	0.86	0.86	126

Pipeline P3.C.R The P3.C.R, the best pipeline that TPOT identified working on the Custom_Pool of elements and on ML_Rolling_Dataset, is reported in Listing 4. Most of the elements used in Listing 4 have already been introduced. The comments should clarify the new pipeline elements. The classification report is shown in Table 5.

```

1 from sklearn.pipeline import make_pipeline
2 from sklearn.naive_bayes import BernoulliNB #Naive
   Bayes classifier for multivariate Bernoulli
   models
3 from sklearn.svm import LinearSVC # Linear SVM
   Classifier
4 from xgboost import XGBClassifier
5 from tpot.builtins import StackingEstimator
6 pipe=make_pipeline(
7     # 1st pipeline element
8     StackingEstimator(
9         estimator=LinearSVC(
10             C=1.0, dual=True, loss="
               squared_hinge",
11             penalty="l2", tol=0.0001)
12             # Linear SVM classifier),
13     # 2nd pipeline element
14     StackingEstimator(
15         estimator=BernoulliNB(alpha=10.0,
16                               fit_prior=True)),
17     # 3rd pipeline element
18     XGBClassifier(
19         learning_rate=0.01, max_depth=3,
20         min_child_weight=12, n_estimators
21         =100, n_jobs=1, subsample
22         =0.950000000000000001,
23         verbosity=0)

```

Listing 4 The P3.C.R Pipeline Code

4.3 Further Analysis on the P3.C.R Pipeline

In this section, we focus on the P3.C.R pipeline described in Sect. 4.2, which is the best-performing pipeline according to the results shown in Table 3.

Training curve To investigate possible overfitting, we randomly extract subsets of the training data, considering sizes ranging from 1 to 100% of the training set. For each subset, we train the pipeline on it, then evaluate on both the test set and the training set, and compute the F1 score in both cases. Plotting these two performances against the subset sizes yields the training curves, shown in Fig. 3. The plot in Fig. 3 shows that a training size of 100 elements is sufficient to achieve peak performance. Moreover, enlarging the dataset does not appear to lead to overfitting, as test performance does not deteriorate significantly.

Sensitivity analysis To assess the stability of the results, we also conducted a sensitivity analysis of the P3.C.R pipeline. The pipeline was trained 100 times on the same dataset (the training subset of ML_Rolling_Dataset), and in each run, performance was evaluated on the test set. The mean and standard deviation of the F1 scores were 0.8202 and 0.0015, respectively.

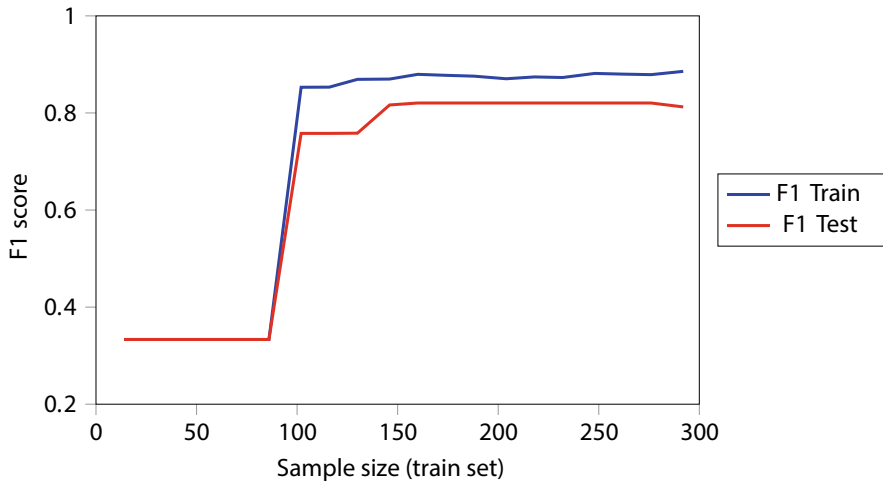


Fig. 3 Pipeline P3.C.R training curve. The x-axis focuses on the train set sample size, while the y-axis focuses on the F1 score performance

4.4 Comparisons

The P3.C.R and P1.C.M pipelines described in Sects. 4.2 and 4.3 respectively have been compared with existing models proposed in the literature. The selected models and the rationale for their choice are outlined below.

The penalized Logit introduced in Pigni (2021) is among the state-of-the-art models for EWS. Since the dataset used in this chapter is balanced, and the penalized Logit is further explored in other chapters of this book, a simple Logistic Regression has been used for comparison with the proposed pipelines. According to Pigni (2021), Casabianca et al. (2022) and Li (2025), economists have increasingly employed supervised ML methods, including Random Forests (RF) and XGBoost. Therefore, the following models have been selected as baselines.

- A Logit Classifier (McCullagh & Nelder, 2019);
- A Random Forest model (Breiman, 2001);
- An XGBoost model (Chen & Guestrin, 2016).

A grid search has been performed to identify the best hyperparameters for each baseline model. The P3.C.R and P1.C.M pipelines were excluded from the grid search, since they are already optimized. The base model optimization was performed twice, both on the ML_Dataset and on the ML_Rolling_Dataset. The best hyperparameters were searched on the Balanced_Train subsets of both datasets using a fivefold cross-validation. The best hyperparameters found are shown in Listings 8, 9, and 10 in Appendix C, while Table 6 reports the pipeline performance. Each pipeline was fitted on the Balanced_Train and evaluated on the Balanced_Test. The

Table 6 F1 score, precision, recall and area under the PR curve for both considered datasets and for each model pipeline; precision and recall values are averaged over the two classes (crisis, non-crisis)

Dataset	Pipeline	F1 score	Precision	Recall	PR_AUC
ML_Dataset	Logit	0.776	0.778	0.778	0.806
ML_Dataset	RF	0.777	0.778	0.778	0.874
ML_Dataset	XGB	0.769	0.770	0.770	0.862
ML_Dataset	P1.C.M	0.785	0.786	0.786	0.854
ML_Rolling_DS	Logit	0.817	0.817	0.817	0.871
ML_Rolling_DS	RF	0.794	0.794	0.794	0.889
ML_Rolling_DS	XGB	0.848	0.849	0.849	0.890
ML_Rolling_DS	P3.C.R	0.856	0.857	0.857	0.892

ML_Rolling_DS is the short for ML_Rolling_Dataset. For each column, the best value for the ML_Dataset and for the Rolling_DS dataset is written in bold.

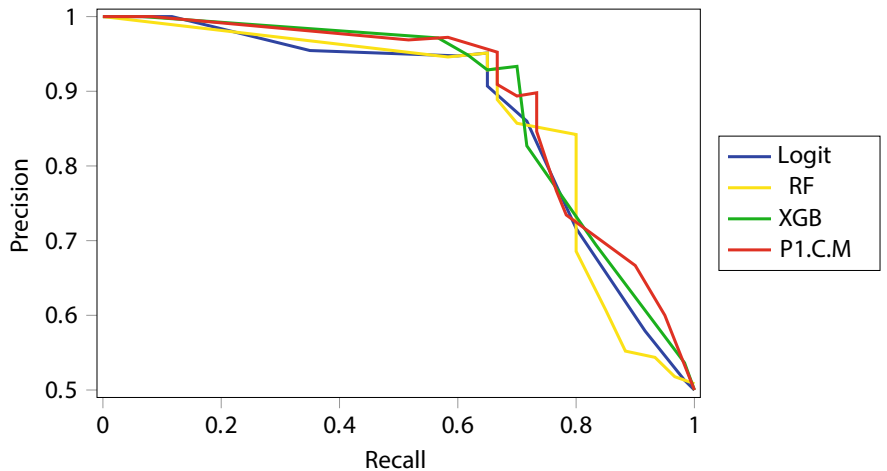


Fig. 4 Precision-Recall curves of the pipelines Logit, RF, XGB, and P1.C.M, using the ML_Dataset

performances are shown in Table 6, which also reports for each pipeline the area under the Precision-Recall (PR) curve. According to Pigni (2021), the PR curve is preferred over the ROC curve when dealing with imbalanced datasets (which is the majority of cases in EWSs). Even though the dataset used in this work is not imbalanced, the PR curve was chosen to facilitate comparisons with other works.

The P1.C.M and P3.C.R pipelines have the best F1 scores (within the context of the ML_Dataset and ML_Rolling_Dataset). It was expected as the result of the optimization described in Sect. 4. More details in Figs. 4 and 5.

Even though the analysis of the pipeline output shows that the P3.C.R and P1.C.M pipelines perform slightly better than the state-of-the-art considered as the reference baseline(s), the performance gain is minimal and falls within the range of variabil-

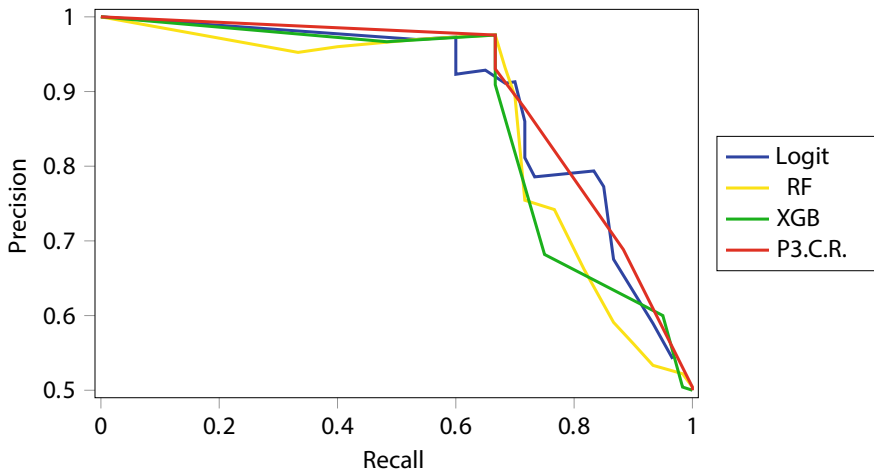


Fig. 5 Precision-Recall curves of the pipelines Logit, RF, XGB, and P3.C.R., using the ML_Rolling_Dataset

ity in the performance (i.e., training the same classifiers multiple times can still yield slightly different performance, a well-known issue in machine learning). Furthermore, the work performed to build the ML_Dataset and ML_Rolling_Dataset undoubtedly had a positive impact on both the proposed approach and the literature reference models. Since most of the literature focuses on imbalanced datasets, this work is not directly comparable to those approaches. Therefore, the performance contribution of Auto ML in the EWSs for financial crises has to be better evaluated in future work.

It is worth noting that the genetic programming optimization performed by the TPOT library may result in a local minimum; therefore, the pipelines identified in this chapter may be further improved. Generally speaking, improving the number of generations or working on a larger dataset can improve the stability of genetic algorithm searches. Unfortunately, in this scenario, it was not possible to further extend the train set (for example, at the expense of the test), because doing so would have made both datasets imbalanced or not anymore representative.

4.5 SHAP Values

The Shapley Additive exPlanations (SHAP) method was used to investigate the pipeline models P1.C.M and P3.C.R presented in Sects. 4.2 and 4.3. This model-agnostic method, based on cooperative game theory, provides consistent and locally accurate feature attributions for individual predictions (Lundberg & Lee, 2017). SHAP values quantify the contribution of each input feature to the model's output by

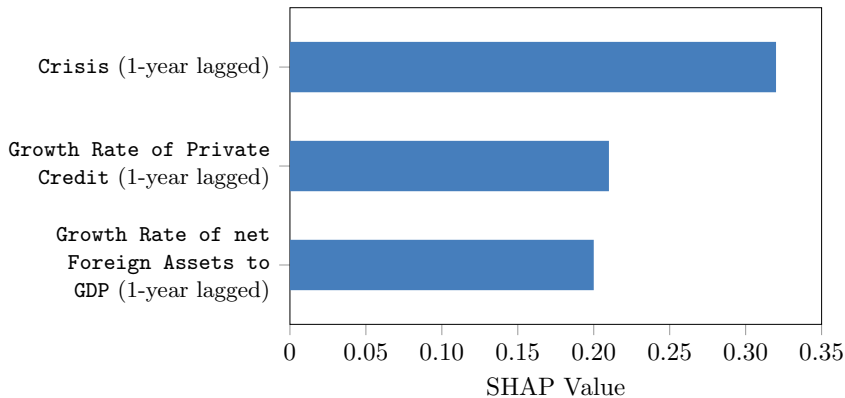


Fig. 6 Barchart showing the Mean Absolute SHAP values computed using the P1.C.M pipeline on the ML_Dataset (the test set was used)

computing the average marginal contribution of a feature across all possible feature subsets. This approach enables both global interpretability (by aggregating feature contributions across the dataset) and local interpretability (by explaining individual predictions), offering a comprehensive understanding of model behavior and feature importance.

Figures 6 and 7 show the mean absolute SHAP values computed using the P1.C.M and P3.C.R pipelines, respectively, on the ML_Dataset and ML_Rolling_Dataset. Specifically, the SHAP values shown were calculated on the test set.

Figure 6 shows the features that contribute most to crises prediction using the pipeline P1.C.M trained on the ML_Dataset; in particular they are

- One-year lagged Crisis. As expected, the occurrence of a crisis in the previous year provides useful information that increases the likelihood of predicting a future crisis.
- One-year lagged Growth Rate of Private Credit. It measures the growth of real domestic credit. It is expected to have a positive effect on the probability of a crisis, which is likely to increase with over-indebtedness and deterioration in the quality of bank assets.
- One-year lagged Growth Rate of net Foreign Assets to GDP. It tracks the evolution of the country's external position. A decline in this ratio is generally expected to increase the probability of a crisis, as it signals widening external imbalances, potential capital flight, or a reduced capacity to absorb external shocks.
- The other features have less evident effects.

Figure 7 shows the features that contribute most to crises prediction using the pipeline P3.C.R trained on the ML_Rolling_Dataset. It is worth recalling that, as outlined in Sect. 3.4, features in this dataset have been enriched with previous years'

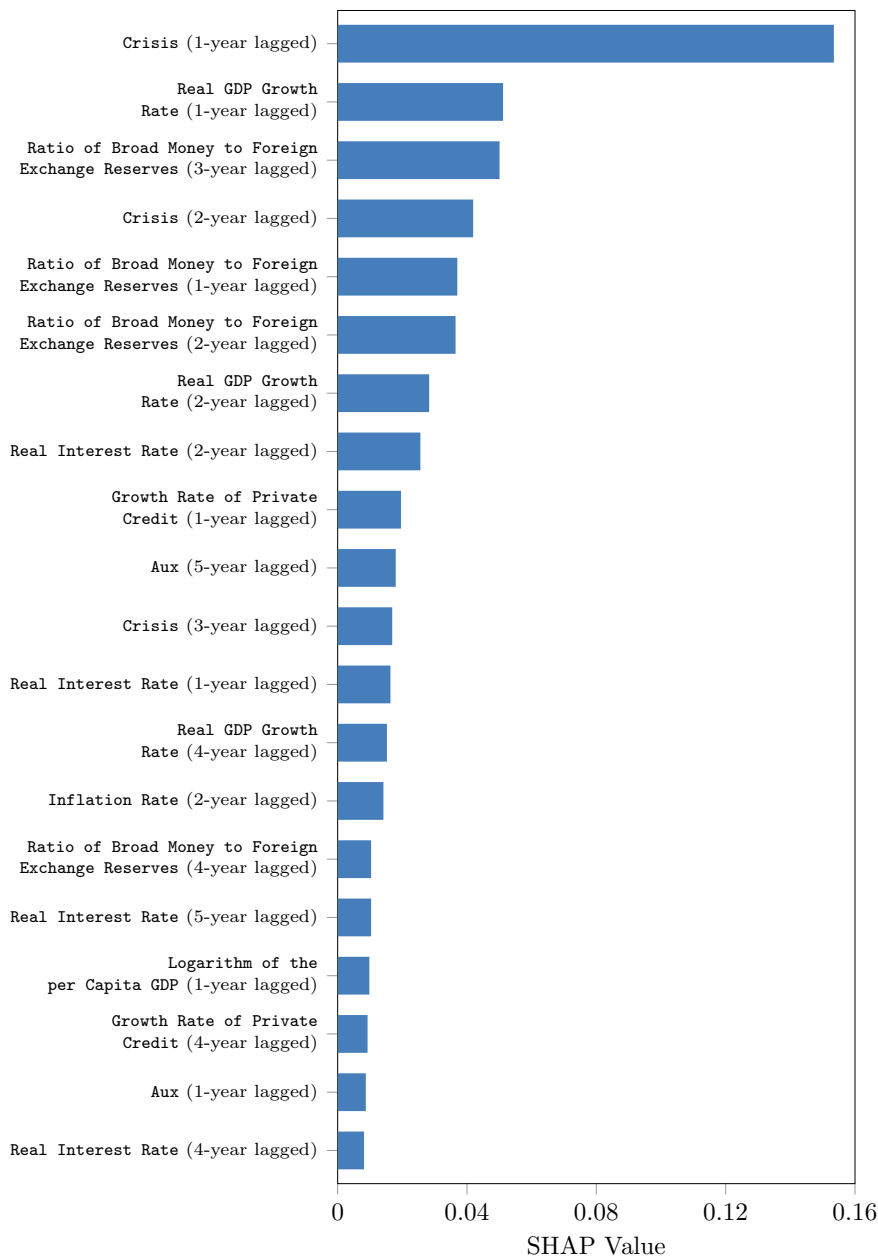


Fig. 7 Barchart showing the Mean Absolute SHAP values computed using the P3.C.R pipeline on the ML_Rolling_Dataset (the test set was used)

values. The most important features, according to the mean absolute SHAP values, are the following (only the first ones are commented on):

- One-year and two-year lagged `Crisis`. As already discussed, past crisis occurrences provide useful information to predict future instabilities.
- One-year lagged `Real GDP Growth Rate`. Lower values of this variable are typically associated with weak macroeconomic conditions and may indicate increasing financial stress.
- Three- and two-year lagged `Ratio of Broad Money to Foreign Exchange Reserves`. These are monetary variables, which, as expected, positively affect the probability of predicting a crisis.

As outlined in Fig. 7, the addition of previous years' data has been taken into account by the classifier. This can explain the (slightly) better performance of the P3.C.R. over the P1.C.M pipeline.

4.6 Considerations

Different runs of TPOT (on the same dataset and element pools) are likely to produce different pipelines for several reasons. For instance, multiple pipelines may achieve comparable top performance (although increasing the number of generations could further differentiate them); most pipeline elements are randomly initialized; and even when a seed is specified, training the same pipeline multiple times can still yield slightly different performance, a well-known issue in machine learning. Nevertheless, this does not represent a significant problem, as the pipelines identified by Auto ML tools are intended to serve as starting points rather than final outcomes of a research effort.

5 Conclusions and Future Work

This work has focused on ML methods to show their potential. Statistical models and ML methods are complementary, often overlapping and enriching each other. Statistical models, such as those presented in different chapters of this book, are based on assumptions and aim to estimate parameters that can be accurately interpreted to explain associations between variables, including inference, while also providing measures of uncertainty through confidence intervals, p-values, and hypothesis tests. Moreover, thanks to these assumptions, they remain stable with small datasets and are robust because their simplicity and constraints often allow them to remain valid with new data, as long as the assumptions hold.

As illustrated in this chapter, the ML approach is oriented toward building predictive models by learning from data without necessarily requiring specific assumptions. ML algorithms, such as those applied here, can capture complex, non-linear

patterns without the user defining an explicit model structure. However, the statistical literature (Bishop, 2006; Giudici, 2003; Hastie et al., 2009; James et al., 2013) documents several limitations—especially for complex models such as neural networks or ensembles (e.g., random forests, XGBoost)—they are often “black boxes”, difficult to interpret, and more focused on prediction than on inference. Non-parametric models, in particular, require large datasets to perform well, are prone to overfitting without proper regularization and validation, and can be less robust in new contexts or when applied to data from different distributions. In addition, some algorithms (e.g., deep learning and boosting) are computationally more demanding to train and validate than simpler models.

Two possible paths, therefore, emerge. One is to choose between the approaches: ML is particularly suitable for complex data structures because it does not require explicit assumptions, whereas when data are limited and accurate, and an explainable interpretation is needed, statistical models are often preferable for their interpretability and robustness. The other path is to combine the two in hybrid approaches. Statistical models provide a solid basis for analysis and interpretation, while ML can refine predictions. For example, statistical models can assess variable importance within an ML framework, or Bayesian ML methods—such as Bayesian optimization and probabilistic graphical models—can integrate both paradigms to improve predictions, especially under uncertainty or incomplete data. Statistical techniques can also prepare data for ML (e.g., normalization, principal component analysis, variable selection), while statistical methods such as Bayesian inference or cross-validation can optimize ML hyperparameters. Applications include health economics, where the goal is to combine the predictive power of ML methods with interpretability (Athey & Imbens, 2017; Chernozhukov et al., 2018).

Appendix A

The algorithm used for undersampling and train-test split is shown in Listing 1.

Appendix B

In this appendix the pipelines not previously shown are reported next.

B.1 Pipeline P1.C.M

The P1.C.M, i.e., the best pipeline that TPOT identified working on the Custom_Pool, of elements and on ML_Dataset, is reported in Listing 5.

Algorithm 1 Undersampling and train-test split using *sel* as stratification variable

```

1: Print "Performing undersampling and train & test split"
2: Print "The initial dataset is stored in the df dataframe"
3: Print "A new variable df_group will be added, accepting the values 'balanced_train', 'balanced_test', 'test2', and 'test3'. The initial default value is 'test3'."
4: Let testSizePerc = 0.3 # Test size is 30%
5: for all group g and frequency f in value_counts of df.sel do
6:   Let df_group ← subset of df where sel == g
7:   Let c1 ← number of rows in df_group where crisis0 == 1
8:   Let c0 ← number of rows in df_group where crisis0 == 0
9:   Let m ← min(c0, c1) # m is the size of the minority class
10:  if m == 0 then # Either the crisis or non-crisis cases are missing
11:    Assign all rows in df_group to group 'test3' # These records would be completely
    dropped by undersampling. They are labelled test3 because they can later be used as an additional
    test set
12:  else if m == 1 then # If the minority class has size 1, it cannot be a train-test split, so it is
    assigned to the train subset
13:    for target in {0, 1} do # in both crisis and non-crisis subsets
14:      Randomly sample 1 row from df_group where crisis0 == target
15:      Assign sampled row to group 'balanced_train'
16:    end for
17:  else # The minority class has at least 2 records (at least one for the train set and one for the
    test set)
18:    Assign all rows in df_group to group 'test2' # test2 is the initial value at the
    beginning
19:    Let t ← round(m · testSizePerc) # t is the test set size of the minority class
20:    Let t ← max(1, t) # if t is lower than 1, then it is set to 1.
21:    Let r ← m − t # train set size on the minority class
22:    for target in {0, 1} do
23:      Randomly sample m rows from df_group where crisis0 == target
24:      Randomly sample t rows from the m rows
25:      Assign the sampled t rows to group 'balanced_test'
26:      Assign the remaining m rows to group 'balanced_train' # All labels assigned
        neither to balanced_test' nor to balanced_train' will keep the initial 'test2' label. At the end of
        the process, these are the records dropped by undersampling. They are labeled test2 because
        they can later be used as an additional test set.
27:    end for
28:  end if
29: end for

```

The `make_pipeline` class creates a pipeline instance. Its parameters are the pipeline elements, in the exact order they will occupy the pipeline. In Listing 5, the pipeline is composed of two elements. The first is the `MakeUnion` component, which creates parallel branches in a pipeline. The same input is passed to different parallel branches, which are defined when the `MakeUnion` class is instantiated. In this case, the `MakeUnion` function has two branches: the first, named `StackingEstimator`, is a class provided by `TPOT` for creating pipelines involving multiple classifiers. It is a meta-transformer that wraps a classifier and adds its predictions and/or class probabilities as synthetic feature(s) to the input dataset. When the `StackingEstimator` is created, a classifier is specified and stored

```

1  from sklearn.pipeline import make_pipeline,
    make_union
2  from sklearn.tree import DecisionTreeClassifier
3  from sklearn.naive_bayes import BernoulliNB
4  from sklearn.preprocessing import
    FunctionTransformer
5  from copy import copy # it returns a shallow copy
    of the object
6  from sklearn.ensemble import ExtraTreesClassifier
7  from tpot.builtins import StackingEstimator
8
9  pipe=make_pipeline(
10     make_union( # 1st pipeline element
11         StackingEstimator(
12             estimator=make_pipeline(
13                 StackingEstimator(
14                     estimator=BernoulliNB(alpha=10.0, fit_prior=
15                         True)),
16                     DecisionTreeClassifier(criterion="gini",
17                         max_depth=6, min_samples_leaf=11,
18                         min_samples_split=11))),
19         FunctionTransformer(copy) # it copies the input
20             features),
21     ExtraTreesClassifier( # 2nd and last pipeline
22         element
23     bootstrap=False, criterion="gini", max_features
24         =0.05, min_samples_leaf=2, min_samples_split
25         =13, n_estimators=100 ))

```

Listing 5 The P1.C.M Pipeline Code

internally. The `StackingEstimator` implements the `fit()`, `transform()`, and `predict()` methods required for pipelines, as outlined in Sect. 2.1. During prediction, the internal classifier processes the input features; its predictions are merged with the original input, and the combined result is passed to the next pipeline element. In our case, the `StackingEstimator` combines the predictions of two classifiers, a `BernoulliNB` and a `DecisionTreeClassifier`.

The second `MakeUnion` branch is the `FunctionTransformer(copy)` class, which creates a copy of the input features. The `MakeUnion` then merges the `DecisionTreeClassifier` results and the input features. Finally, the second pipeline element is the `ExtraTreesClassifier`, which uses the merge to produce the final prediction. The entire pipeline is trained on the training set and evaluated on the test set. The classification report obtained is shown in Table 7.

The use of `StackingEstimator` and `FunctionTransformer(copy)` inside the `make_union` produces a `make_union` output where the original input features are repeated twice. In fact, the `StackingEstimator` output contains the input features, and the `FunctionTransformer(copy)` copies the input

Table 7 Classification report of the pipeline P1.C.M. evaluated on the test set

	Precision	Recall	F1 score	Support
Non-crises (0)	0.76	0.83	0.79	63
Crises (1)	0.81	0.75	0.78	63
Accuracy	–	–	0.79	126
Average	0.79	0.79	0.79	126

Table 8 Classification report of the pipeline P2.L.M evaluated on the test set

	Precision	Recall	F1 score	Support
Non-crises (0)	0.74	0.90	0.81	63
Crises (1)	0.88	0.68	0.77	63
Accuracy	–	–	0.79	126
Average	0.81	0.79	0.79	126

features. Therefore, it is wise to manually restructure the pipeline as outlined in Listing 3.

The performance between P1.C.M and P1.C.M.Bis does not change much. However, tuning the last element (`ExtraTreesClassifier`) of the P1.C.M.Bis pipeline while keeping the preceding components unchanged could potentially improve classification performance. This evaluation will be considered in future studies.

B.2 Pipeline P2.L.M

The P2.L.M, i.e., the best pipeline that TPOT identified working on the `Light_Pool` of elements and on `ML_Dataset` is reported in Listing 6. Most of the elements used in Listing 6 have already been introduced in Appendix B.1. The comments should clarify how the new pipeline elements work. The classification report is shown in Table 8.

B.3 Pipeline P4.L.R

The P4.L.R, i.e., the best pipeline that TPOT identified working on the `Light_Pool` of elements and on `ML_Rolling_Dataset` is reported in Listing 7. Most of the elements used in Listing 7 have already been introduced in Appendix B.1. The comments should clarify the new pipeline elements. The classification report is shown in Table 9.

```
1 from sklearn.pipeline import make_pipeline,
  make_union
2 from sklearn.preprocessing import
  FunctionTransformer
3 from sklearn.feature_selection import
  SelectPercentile
4 from sklearn.feature_selection import f_classif #
  # Compute the ANOVA F-value for the provided
  sample
5 from sklearn.preprocessing import Binarizer
6 from sklearn.naive_bayes import GaussianNB
7 from sklearn.neighbors import KNeighborsClassifier
8 from copy import copy
9
10 pipe=make_pipeline(
11 make_union( # 1st pipeline element
12 FunctionTransformer(copy),
13 make_pipeline(
14 StackingEstimator(estimator=GaussianNB()),
15 SelectPercentile( # Select features according to a
  percentile of the highest scores
16 score_func=f_classif, # see import
17 percentile=45
18 )
19 )
20 ),
21 # 2nd pipeline element
22 Binarizer(threshold=0.25), # Binarize data (set
  feature values to 0 or 1) according to a
  threshold.
23 # 3rd and last pipeline element
24 KNeighborsClassifier(n_neighbors=13, p=2, weights=
  "distance")
25 )
```

Listing 6 The P2.L.M Pipeline Code

Table 9 Classification report of the pipeline P4.L.R evaluated on the test set

	Precision	Recall	F1 score	Support
Non-crises (0)	0.81	0.92	0.86	63
Crises (1)	0.91	0.78	0.84	63
Accuracy	–	–	0.85	126
Average	0.86	0.85	0.85	126

Appendix C

The (literature) baseline models described in Sect. 4.4 and the optimal hyperparameters found are outlined in Listings 8, 9, and 10.

```

1 from sklearn.pipeline import make_pipeline
2 from sklearn.preprocessing import Binarizer
3 from sklearn.tree import DecisionTreeClassifier
4
5 pipe=make_pipeline(
6     Binarizer(threshold=0.45),
7     DecisionTreeClassifier(criterion="gini",
8         max_depth=4, min_samples_leaf=5,
9         min_samples_split=4)
10 )

```

Listing 7 The P4.L.R Pipeline Code

```

1 logitPipeline = Pipeline([
2     ('logreg', LogisticRegression(
3         max_iter=5000,
4         C=0.1,
5         penalty='l1',
6         solver='liblinear'
7     ))
8 ])

```

Listing 8 A baseline pipeline featuring a Logistic Regression classifier. The hyperparameters were selected via a grid search across both ML_Dataset and ML_Rolling_Dataset. The two different searches produced the same hyperparameters

```

1 rfPipe = Pipeline([
2     ('clsf', RandomForestClassifier(
3         bootstrap=True,
4         max_depth=None, #=10 for the rolling dataset
5         min_samples_leaf=1,
6         min_samples_split=2,
7         n_estimators=100
8     ))
9 ])

```

Listing 9 A baseline pipeline featuring a Random Forest classifier. The hyperparameters were selected via a grid search across both ML_Dataset and ML_Rolling_Dataset. The two different searches produced similar but not equal hyperparameters. The differences are outlined using comments

```

1 xgbPipe = xgb.XGBClassifier(
2   objective='binary:logistic',
3   use_label_encoder=False,
4   eval_metric='logloss',
5   colsample_bytree=0.8,
6   gamma=1, # =0 for the rolling dataset
7   learning_rate=0.01,
8   max_depth=3,
9   min_child_weight=1, # =5 for the rolling dataset
10  n_estimators=100,
11  subsample=1.0      # = 0.8 for the rolling dataset
12 )

```

Listing 10 An XGBoost classifier. The hyperparameters were selected via a grid search across both ML_Dataset and ML_Rolling_Dataset. The two different searches produced similar but not equal hyperparameters. The differences are outlined using comments

References

- Athey, S., & Imbens, G. (2017). The state of applied econometrics: Causality and policy evaluation. *Journal of Economic Perspectives*, 31, 3–32.
- Azedou, A., Amine, A., Kisekka, I., & Lahssini, S. (2025). Genetic algorithm optimization of ensemble learning approach for improved land cover and land use mapping: Application to talassemiane national park. *Ecological Indicators*, 177, 1–14.
- Azure. (2025). Automated machine learning. <https://azure.microsoft.com/en-us/solutions/automated-machine-learning>. Retrieved August 20, 2025
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. New York: Springer.
- Breiman, L. (2001). Random forests. *Machine learning*, 45, 5–32.
- Casabianca, E. J., Catalano, M., Forni, L., Giarda, E., & Passeri, S. (2022). A machine learning approach to rank the determinants of banking crises over time and across countries. *Journal of International Money and Finance*, 129, 1–24.
- Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16, 321–357.
- Chen, T., & Guestrin, C. (2016). XGboost: A scalable tree boosting system. In C. Aggarwal, A. Smola, R. Rastogi, D. Shen, B. Krishnapuram, & M. Shah (Eds.), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16* (pp. 785–794). Association for Computing Machinery.
- Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W., & Robins, J. (2018). Double/debiased machine learning for treatment and structural parameters. *The Econometrics Journal*, 21, C1–C68.
- Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55, 78–87.
- EpistasisLab (2025). Tpot documentation. <https://github.com/EpistasisLab/tpot>. Retrieved August 20, 2025.
- Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., & Hutter, F. (2015). Efficient and robust automated machine learning. *Advances in Neural Information Processing Systems*, 28, 2962–2970.
- Giudici, P. (2003). *Applied data mining: Statistical methods for business and industry*. Wiley.
- Google Cloud (2025). *Cloud AutoML*. <https://cloud.google.com/automl>. Accessed 20 Aug, 2025..
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning. Data mining, inference, and prediction* (2nd ed.). Springer.

- He, H., & Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21, 1263–1284.
- He, X., Zhao, K., & Chu, X. (2021). AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212, 1–27.
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning*. New York: Springer.
- Jin, H., Song, Q., & Hu, X. (2019). Auto-keras: An efficient neural architecture search system. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1946–1956).
- Krawczyk, B. (2016). Learning from imbalanced data: Open challenges and future directions. *Progress in Artificial Intelligence*, 5, 221–232.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521, 436–444.
- Leung, F.H.-F., Lam, H.-K., Ling, S.-H., & Tam, P.K.-S. (2003). Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural Networks*, 14, 79–88.
- Li, L. (2025). Research on the construction of international financial crisis early warning model based on xgboost algorithm. In M. Zhang, N. A. Y. Hui, A. T. Rahman, & K. Shehzad (Eds.), *Proceedings of the 2nd Guangdong-Hong Kong-Macao Greater Bay Area International Conference on Digital Economy and Artificial Intelligence, DEAI '25* (pp. 1335–1339). Association for Computing Machinery.
- Lipton, Z. C. (2018). The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16, 31–57.
- Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, 30, 4765–4774.
- Mahalle, A. V., et al. (2024). Machine learning algorithms for data-driven intelligent systems. In M. Ghonghe, et al. (Eds.), *Data-driven systems and intelligent applications* (pp. 52–61). Taylor & Francis.
- McCullagh, P., & Nelder, J. A. (2019). *Generalized linear models*. Boca Raton, FL: Routledge.
- Mitchell, M. (1998). *An introduction to genetic algorithms*. Cambridge, MA: MIT Press.
- Nazareth, N., & Reddy, Y. V. R. (2023). Financial applications of machine learning: A literature review. *Expert Systems with Applications*, 219, Article 119640.
- Nielsen, M. A. (2015). *Neural networks and deep learning*. San Francisco, CA: Determination Press.
- Olson, R. S., & Moore, J. H. (2016, 24 Jun). TPOT: A tree-based pipeline optimization tool for automating machine learning. In F. Hutter, L. Kotthoff, & J. Vanschoren (Eds.), *Proceedings of the Workshop on Automatic Machine Learning, Volume 64 of Proceedings of Machine Learning Research* (pp. 66–74). PMLR.
- Olson, R. S., et al. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference, 2016*, 485–492.
- Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12, 2825–2830.
- Perez, E., Kiela, D., & Cho, K. (2021). True few-shot learning with language models. *Advances in Neural Information Processing Systems*, 34, 11054–11070.
- Pigini, C. (2021). Penalized maximum likelihood estimation of logit-based early warning systems. *International Journal of Forecasting*, 37, 1156–1172.
- Python Software Foundation. (2025). The python language reference. <https://docs.python.org/3/reference/index.html>. Retrieved August 20, 2025.
- Qcut. (2025). Quantile based discretization function. <https://pandas.pydata.org/docs/reference/api/pandas.qcut.html>. Retrieved August 20, 2025.
- Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 33, pp. 4780–4789).

- Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61, 85–117.
- Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, 25, 1–9.
- van Rossum, G., & Drake, F. L. (2011). *The python language reference manual*. Network Theory Ltd.
- Yaloveha, V., Podorozhniak, A., & Kuchuk, H. (2022). Convolutional neural network hyperparameter optimization applied to land cover classification. *Radioelectronic and computer systems*, 101, 115–128.
- Zoph, B., & Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *International Conference on Learning Representations* (pp. 1–16).

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

